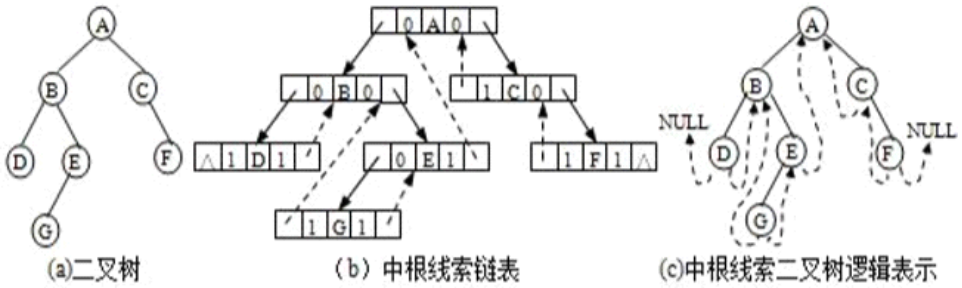
线索、森林、赫夫曼、还原

2019年2月9日 21:43

-
- 线索二叉树

1. 线索二叉树：为每个结点加上线索的二叉树称为线索二叉树
   1) 线索：指向前驱结点和后继结点的指针称为线索
   2) 线索化：对二叉树进行某种次序遍历使其变为线索二叉树的过程
      i. 实质为将空指针改为指向前驱或后继的线索，并置对应tag=1
   3) typedef struct thrNode{
      ElemType data;
      struct thrNode *Lchild, *Rchild; /* 左右指针*/
      int Ltag, Rtag;
      /*左右指针类型标志,0表示指针，1表示线索*/
      } *ThrNode;
   4) 

2. 中序遍历线索化二叉树
   1) 全局变量pre指向前一个访问的结点，即当前的t的前驱
   2) void In_thread (ThrNode *t) { /*中根遍历线索化二叉树t */
      if (t) {
      In_thread (t->lchild) ; /* 左子树线索化 */
      if (t->lchild = = NULL) { /* 建立 t 结点的前驱线索 */
      t->Ltag =1;
      t->lchild = pre; }
      if (pre!=NULL) {
      if (pre->rchild = = NULL) { /* 建立pre结点的后继线索 */
      pre->Rtag = 1;
      pre->rchild = t; } }
      pre = t;
      In_thread (t->rchild); /* 右子树线索化 */ } }
   3) void Crt_thread (ThrNode *t) { /*建立中根线索树 t */
      pre=NULL;
      In_thread (t) ; /* 中根线索化二叉树 */
      pre->Rtag=1;} /* 最后一个结点线索化 */

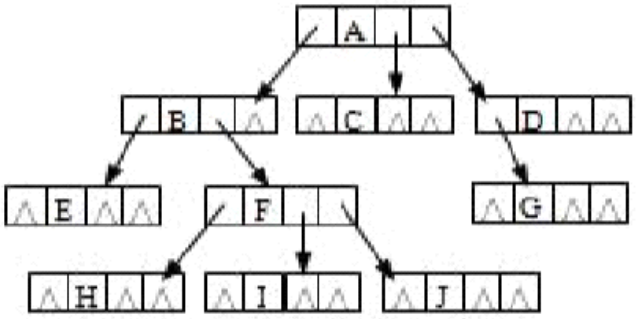
3. 中序遍历线索二叉树
   1) 中根线索树上，遍历序列的第一个结点是线索树上唯一左指针域为空的结点，而序列的最后一个结点是线索树上唯一一个右指针域为空的结点
   2) void InOrder_th(ThrNode *t) { //中序遍历线索二叉树

```
p = t;
if (p!=NULL) {
        while (p->lchild != NULL)
                p = p->lchild;        //找遍历的第一个结点
        printf("%d",p->data); /*访问其左子树为空的结点*/
        while ( p->rchild!=NULL) {  //求结点的后继
                if(p->Rtag==1)
                        p = p->rchild;
                else  {
                        p = p->rchild;
                        while(p->Ltag!=1)
                                p = p->lchild; }
                printf("%d",p->data);  /*访问“右线索”所指后继结点*/  } } }
```

-
- 其他表示法

1. 双亲数组表示法：利用每个结点（除根结点外）只有唯一双亲的特点，用一维数组来存储一棵一般的树
2. 结点定长的孩子链表表示法：其结点结构为：一个数据域和三个指针域。指针域用于指向该结点的各个孩子结点。该树的三重链表如图所示

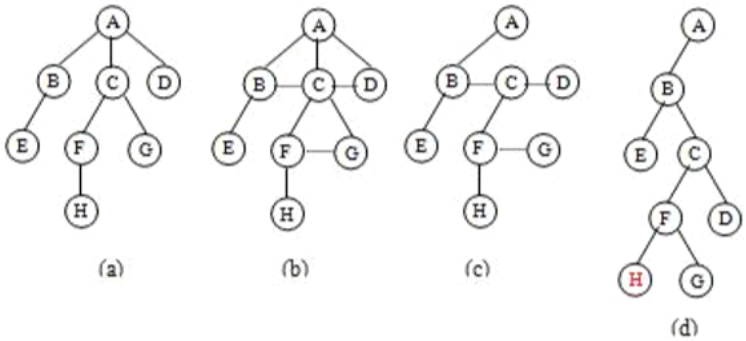
1)



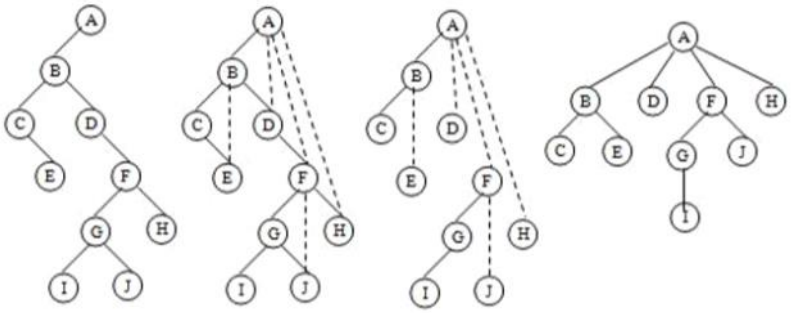
3. 孩子一兄弟二叉链表表示法：其结点结构为：一个数据域和两个指针域。其中一个指针指向它的第一个孩子结点，另一个指向它的兄弟结点

-
- 转换

4. 一般树转换为二叉树
    1) 加线：在各兄弟结点间用虚线相连。可理解为每个结点的兄弟指针指向它的一个兄弟
    2) 抹线：对每个结点仅保留它与其最左一个孩子的连线，抹去该结点与其它孩子之间的连线。可理解为每个结点仅有一个孩子指针，让它指向自己的第一个孩子
    3) 旋转：把虚线改为实线从水平方向向下旋转45°，成右斜下方向。原树中实线成左斜下方向。这样就形成一棵二叉树
    4) 转化成二叉树后，根节点一定没有右子树

5) 

5. 转化前的先根遍历和转化后的先根遍历是相同的，转化前的后根遍历和转化成二叉树后的中根序列是相同的
6. 二叉树还原为一般树
   1) 加线：若某结点i是双亲结点的左孩子，则将该结点i的右孩子以及当且仅当连续地沿着右孩子的右链不断搜索到所有右孩子，都分别与结点i的双亲结点用虚线连接
   2) 抹线：把原二叉树中所有双亲结点与其右孩子的连线抹去。这里的右孩子实质上是原一般树中结点的兄弟，抹去的连线是兄弟间的关系
   3) 进行整理：把虚线改为实线，把结点按层次排列

4) 

7. 森林（树的有限集合）转换为二叉树的步骤为：
   1) 将森林中每棵子树转换成相应的二叉树，形成有若干二叉树的森林
   2) 按森林中树的先后次序，依次将后边一棵二叉树作为前边一棵二叉树根结点的右子树，这样整个森林就生成了一棵二叉树（第一棵树的根结点便是生成后的二叉树的根结点）
8. 二叉树还原为森林的步骤是：
   1) 抹线：将二叉树的根结点与其右孩子的连线以及当且仅当连续地沿着右链不断地搜索到的所有右孩子的连线全部抹去，这样就得到包含若干棵二叉树的森林
   2) 还原：将每棵二叉树按二叉树还原为一般树的方法还原为一般树，得到森林
9. 遍历森林
   1) 先序遍历森林：若森林非空，可按下述规则遍历：
      i. 访问森林中第一棵树的根结点
      ii. 先序遍历第一棵树中根结点的子树森林
      iii. 先序遍历除去第一棵树之后剩余的树构成的森林
   2) 中序遍历森林：若森林非空，则可按下述规则遍历之：
      i. 中序遍历森林中第一棵树的根结点的子树森林
      ii. 访问第一棵树的根结点
      iii. 中序遍历除去第一棵树之后剩余的树构成的森林

◆

- 赫夫曼树

1. 路径：由从树的根结点到其余结点的分支构成一条路径
   1) 路径长度：路径上的分支数目
   2) 树的路径长度WPL：从树根到树中其余每个结点的路径长度之和
   3) 结点的带权路径：从根到带权结点之间的路径长度与结点的权值的乘积
   4) 树的带权路径WPL：树中所有带权结点的带权路径长度之和
2. 赫夫曼树/最优树/最优二叉树：带权路径最短的树
   1) 假设有 n 个权值{w1，w2，...wn }，试构造一棵有 n 个叶子结点的二叉树，每个叶子结点带权为wi。显然，这样的二叉树可以构造出多棵，其中必存在一棵带权路径长度 WPL 取最小的二叉树，称该二叉树为最优二叉树
3. 赫夫曼算法：
   1) 根据给定的n个权值{W1,W2,...,Wn}构成n棵二叉树的集合F={T1,T2,...,Tn},其中每棵二叉树Ti中只有一个带权为Wi的根结点，其左右子树均空
   2) 在F中选取两棵根结点的权值最小的树作为左右子树构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和
   3) 在F中删除这两棵树，同时将新得到的二叉树加入F中
   4) 重复2）3），直到F只含一棵树为止
4. 赫夫曼编码
   1) 前缀编码：设计长短不等的编码，必须是任一个字符的编码都不是另一个字符的编码的前缀的编码
   2) 约定左分支表示字符‘0’，右分支表示字符‘1’，则以由从根到叶子的路径上的分支表示的字符组成的字符串作为该叶子结点字符的编码
   3) 若以字符出现的次数为权，构造一棵赫夫曼树，由此得到的二进制前缀编码便为“最优前缀编码”(赫夫曼编码)

-
- 还原二叉树

1. 根据后序和中序遍历，输出按层遍历（和先序遍历）
   1) 引：对同一棵树，左/右子树的区间长度在不同遍历中不会变
   2) 引：在中序遍历中找到后序遍历末元素下标k，0~k和k+1~末即为中序遍历上的左右子树；0~k和k~末-1即为后序遍历的左右子树
   3) 转：除了最左子树外，区间位置会变，最简单的方法是传址给函数
   4) void dfs(int*post,int*in,int r,int n){
   //参数post和in是当前子树所在区间的起始地址
   //r=树根在后序遍历的相对下标，n=层数
   if(r<0) //没有这个下标，即搜索到终点了
   return;
   l=max(l,n); //更新总层数
   int root=post[r]; //后序的最后一数一定是根
   //cout<<root<<' '; //要求先序遍历的话在这输出或尾插即可
   q[n].push(root);
   int k=0; //k=中序遍历中根的下标

```
        while(in[k]!=root)
            ++k;
        dfs(post,in,k-1,n+1);
            /*既是左子树在中序的最后一个下标，又是左子树
            在后序的最后一个下标，即右子树根在后序的下标*/
        dfs(post+k,in+k+1,r-1-k,n+1);
            /*r-1是右子树在后序的最后一个下标，即右子树根
            右子树区间位置的相对变化：中序+=k+1，后序+=k*/
    }
```

2. 根据先序和中序遍历，输出后序遍历
    1) 引：对同一棵树，左/右子树的区间长度在不同遍历中不会变
    2) 引：在中序遍历中找到先序遍历首元素下标k，0~k和k+1~末即为中序遍历上的左右子树；1~k+1和k+1~末即为先序遍历的左右子树
    3) 转：除了最左子树外，区间位置会变，最简单的方法是传址给函数
    4) 递归终点不太好找，可以类似中后序，把区间长度传给函数
    5)

```
void dfs(int*pre,int*in,int len){
//参数pre和in是当前子树所在区间的起始地址，len是区间长度
    if(len<1)           //没有这个区间，即搜索到终点了
        return;
    int root=pre[0]; //先序的首元素一定是根
    int k=0;            //k=中序遍历中根的下标
    while(in[k]!=root)
        ++k;
    dfs(pre+1,in,k);
                //先序根下标位置+1即为左子树区间起始下标
    dfs(pre+k+1,in+k+1,len-k-1);
                //中序根下标位置+1即为右子树区间起始下标
    v.push_back(root);      //最后遍历到根
}
```

    i.
    ii.
    iii.
    iv.
    v. --------我是底线----------